UNIT IV

Unit-4 UML

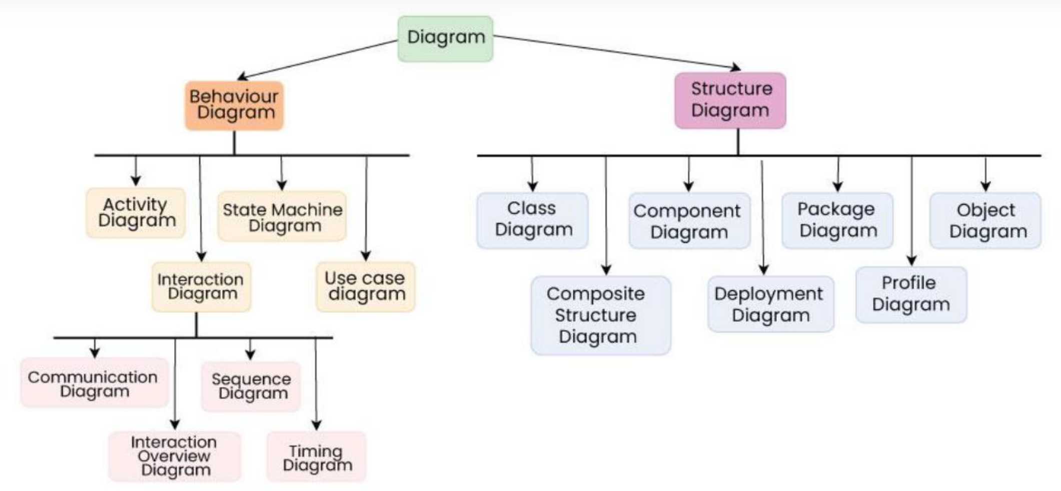
- Fundamental of UML – Associations, Multiplicity, Qualified Association,
- Reflexive Association, Inheritance & Generalization, Dependencies
- Component of UML – Class Diagram, Object Diagram, Use Case Diagram, Activity Diagram
- Case study –Library management system, ticket reservation system, hospital management system

Fundamental of UML – Associations, Multiplicity, Qualified Association

- Unified Modeling Language (UML) is a general-purpose modeling language. The main aim of UML is to define a standard way to visualize the way a system has been designed.
- It is quite similar to blueprints used in other fields of engineering. UML is not a programming language, it is rather a visual language.
- We use UML diagrams to show the behavior and structure of a system.
- UML helps software engineers, businessmen, and system architects with modeling, design, and analysis.
- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.
- Businessmen do not understand code. So UML becomes essential to communicate with non-programmers about essential requirements, functionalities, and processes of the system.

Types of UML Diagrams

UML is linked with object-oriented design and analysis. UML makes use of elements and forms associations between them to form diagrams. Diagrams in UML can be broadly classified as:



1. Association

- An association represents a bi-directional relationship between two classes. It indicates that instances of one class are connected to instances of another class.

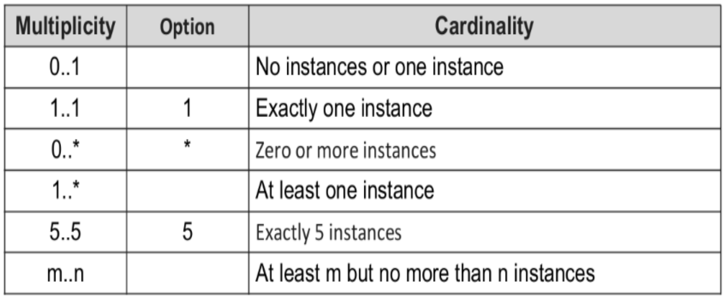
- Associations are typically depicted as a solid line connecting the classes, with optional arrows indicating the direction of the relationship. It shows a "uses-a" or "works-with" relationship.

Example : In a Library Management System:

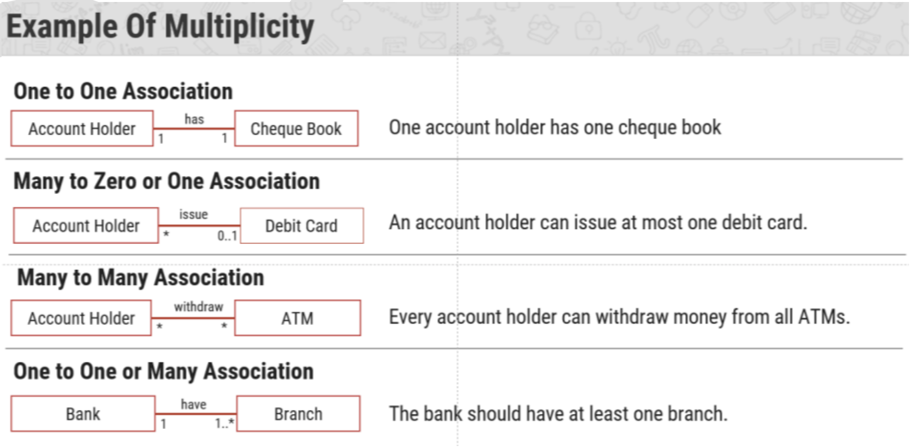
A Student borrows a Book. Association: Student — borrows —> Book

2. Multiplicity

- multiplicity defines the numerical range of instances that can participate in an association between two entities in a Unified Modeling Language (UML) diagram.
- It uses a lower-bound and an upper-bound to specify how many objects of one class are related to one object of another.
- Multiplicity is expressed using numbers, ranges (e.g., 1..5), and the asterisk * to denote an unlimited quantity, ensuring clear communication and accurate system modeling.
- Multiplicity helps in database design (e.g., deciding 1-to-1, 1-to-many, many-to-many relationships).



Multiplicity	Option	Cardinality
0..1		No instances or one instance
1..1	1	Exactly one instance
0..*	*	Zero or more instances
1..*		At least one instance
5..5	5	Exactly 5 instances
m..n		At least m but no more than n instances



Example Of Multiplicity

One to One Association

One account holder has one cheque book

Many to Zero or One Association

An account holder can issue at most one debit card.

Many to Many Association

Every account holder can withdraw money from all ATMs.

One to One or Many Association

The bank should have at least one branch.

Example

1. **Teacher (1) ———————— (0..*) Student**
   One teacher can teach many students.

2. A **Bank Account (1)** belongs to **1..2 Holders.**
   BankAccount (1) ———————— (1..2) AccountHolder
   An account may have one or two holders (joint account).

3. **Qualified association**

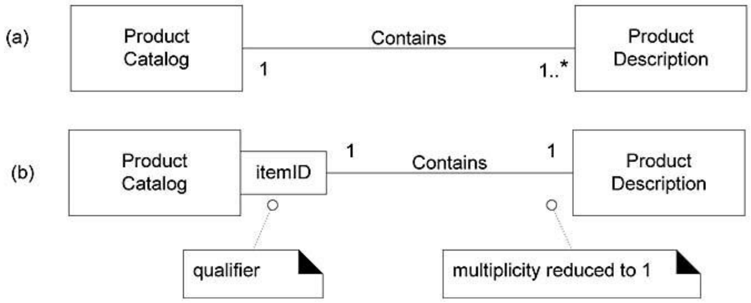
   A **qualified association** in software engineering uses a special attribute called a **qualifier** to select a unique object from a larger, one-to-many or many-to-many relationship. It reduces the multiplicity by introducing a filter. Small rectangle (qualifier) on the association line.

   Example :

   1. Library – Book (via ISBN)

   - A library has thousands of books.
   - To find a particular book, you use ISBN as a qualifier.
   - UML Notation:
   - Library —— [ISBN] —— Book

   2. **A ProductCatalog has many ProductDescription objects. An Item ID can be the qualifier to retrieve a specific product description from the catalog.**



- **Reflexive Association, Inheritance & Generalization, Dependencies**

- **Reflexive Association**

In software engineering, a reflexive association, also known as a self-association, describes a relationship between instances of the same class. In a UML class diagram, this is shown as a line looping back to the same class.

In other word, A relationship within a class where one or more instances of that same class are linked to each other.

- An association where a class is related to itself.
- A line from a class back to itself (often curved).
- Objects of the same class are linked with each other.

Example

- **Employee supervises Employee**.
  - An employee can be a supervisor of another employee.
- Employee ——— supervises ———> Employee (same class)

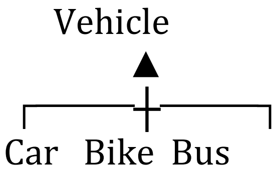
- **Inheritance & Generalization**

Inheritance is the mechanism by which a specialized class (subclass or child class) automatically acquires the attributes and behaviors (methods) of its superclass.

Generalization is the process of identifying common characteristics (attributes and behaviors) among multiple classes and abstracting them into a more general, higher-level class, known as a superclass or parent class. It represents an "is-a" relationship, where the specialized classes "are a type of" the general class.
Example

- Vehicle (general class) → Car, Bike, Bus (specialized classes).

  - All vehicles have attributes like speed, fuel, and operations like move().
  - But Car, Bike, and Bus have their own specializations.



Dependencies

A weaker relationship where one class (client) depends on another (supplier) for some functionality. A dashed arrow from the dependent (client) class to the supplier class. If the supplier class changes, the client class may be affected.

Example:

- **A Student class depends on Library class to borrow books.**
- **Student ─ - - - - - > Library**
- **Here, Student is the client, Library is the supplier.**

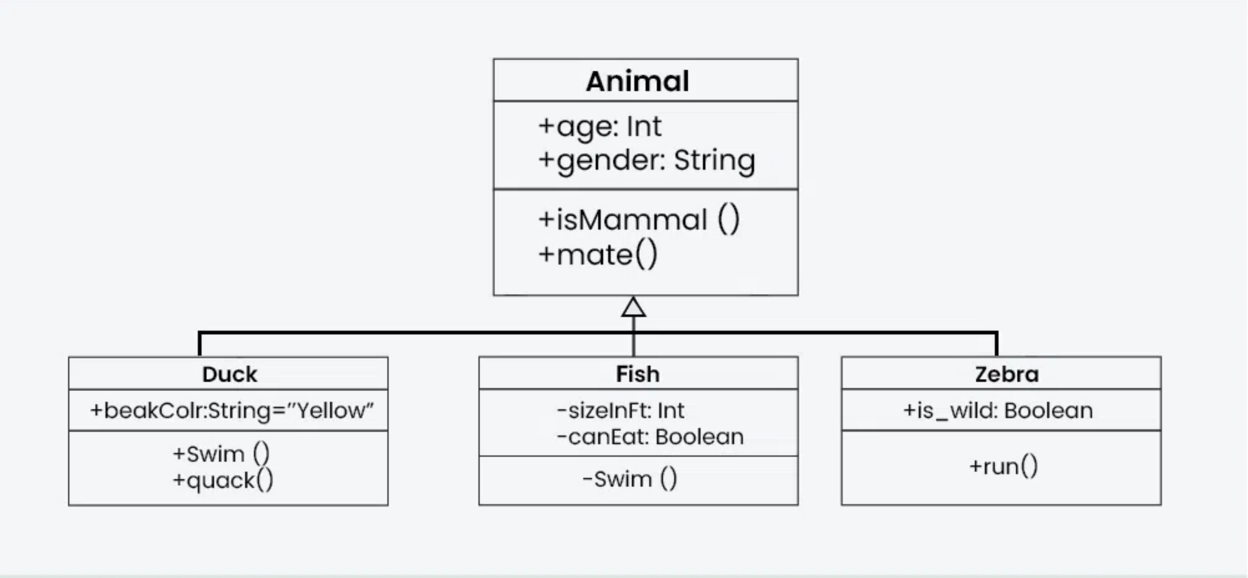
➢ **Component of UML – Class Diagram, Object Diagram, Use Case Diagram, Activity Diagram**

What are class Diagrams?

Class diagrams are a type of UML (Unified Modeling Language) diagram used in software engineering to visually represent the structure and relationships of classes within a system i.e. used to construct and visualize object-oriented systems.

In these diagrams, classes are depicted as boxes, each containing three compartments for the class name, attributes, and methods. Lines connecting classes illustrate associations, showing relationships such as one-to-one or one-to-many.

Class diagrams provide a high-level overview of a system's design, helping to communicate and document the structure of the software. They are a fundamental tool in object-oriented design and play a crucial role in the software development lifecycle.

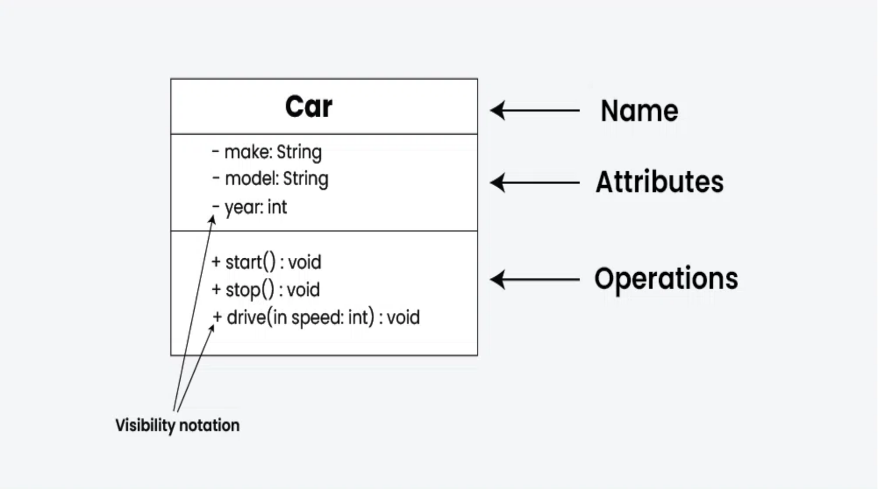


What is a class?

In object-oriented programming (OOP), a class is a blueprint or template for creating objects. Objects are instances of classes, and each class defines a set of attributes (data members) and methods (functions or procedures) that the objects created from that class will possess. The attributes represent the characteristics or properties of the object, while the methods define the behaviors or actions that the object can perform.

UML Class Notation

class notation is a graphical representation used to depict classes and their relationships in object-oriented modeling.

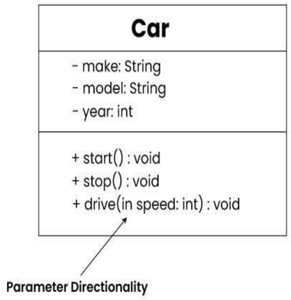


1. **Class Name:**
   - The name of the class is typically written in the top compartment of the class box and is centered and bold.
2. **Attributes:**
   - Attributes, also known as properties or fields, represent the data members of the class. They are listed in the second compartment of the class box and often include the visibility (e.g., public, private) and the data type of each attribute.
3. **Methods:**
   - Methods, also known as functions or operations, represent the behavior or functionality of the class. They are listed in the third compartment of the class box and include the visibility (e.g., public, private), return type, and parameters of each method.
4. **Visibility Notation:**
   - Visibility notations indicate the access level of attributes and methods. Common visibility notations include:
     - + for public (visible to all classes)
     - - for private (visible only within the class)
     - # for protected (visible to subclasses)

    - ~ for package or default visibility (visible to classes in the same package)

Parameter Directionality

In class diagrams, parameter directionality refers to the indication of the flow of information between classes through method parameters. It helps to specify whether a parameter is an input, an output, or both. This information is crucial for understanding how data is passed between objects during method calls.

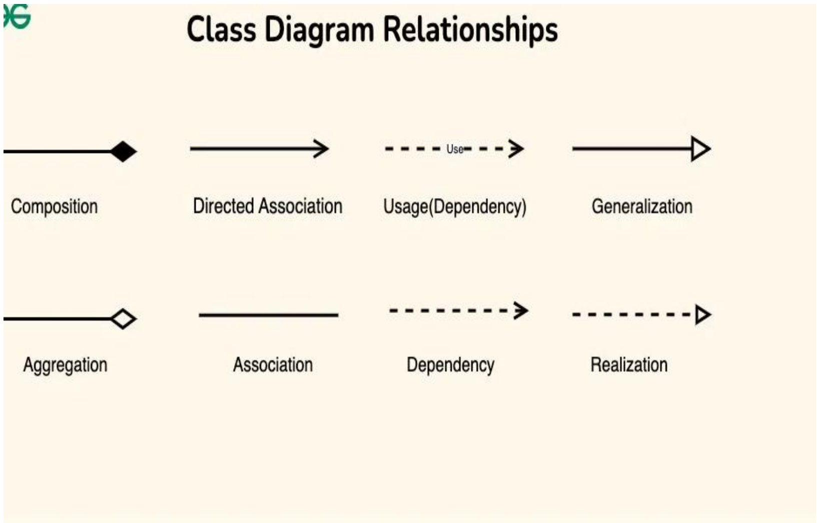


There are three main parameter directionality notations used in class diagrams:

- **In (Input):**
    - An input parameter is a parameter passed from the calling object (client) to the called object (server) during a method invocation.
    - It is represented by an arrow pointing towards the receiving class (the class that owns the method).
- **Out (Output):**
    - An output parameter is a parameter passed from the called object (server) back to the calling object (client) after the method execution.
    - It is represented by an arrow pointing away from the receiving class.
- **InOut (Input and Output):**
    - An InOut parameter serves as both input and output. It carries information from the calling object to the called object and vice versa.
    - It is represented by an arrow pointing towards and away from the receiving class.

Relationships between classes

In class diagrams, relationships between classes describe how classes are connected or interact with each other within a system. There are several types of relationships in object-oriented modeling, each serving a specific purpose. Here are some common types of relationships in class diagrams:



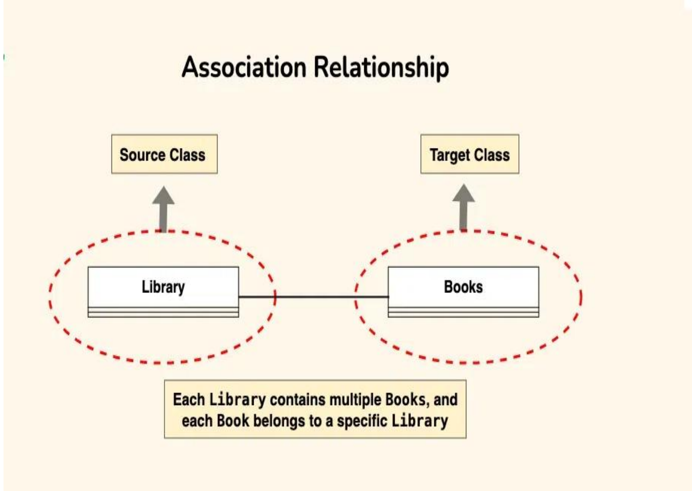
1. Association

An association represents a bi-directional relationship between two classes. It indicates that instances of one class are connected to instances of another class. Associations are typically depicted as a solid line connecting the classes, with optional arrows indicating the direction of the relationship.

Let's understand association using an example:

Let's consider a simple system for managing a library. In this system, we have two main entities: Book and Library. Each Library contains multiple Books, and each Book belongs to a specific Library. This relationship between Library and Book represents an association.

The "Library" class can be considered the source class because it contains a reference to multiple instances of the "Book" class. The "Book" class would be considered the target class because it belongs to a specific library.


Association Relationship
Source Class
Target Class
Library
Books
Each Library contains multiple Books, and each Book belongs to a specific Library

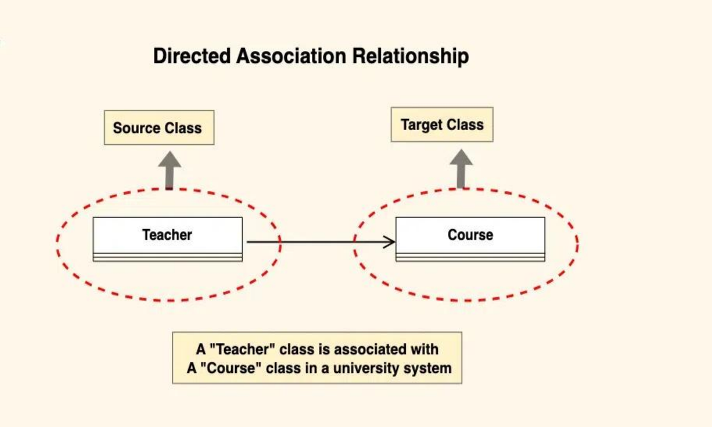
2. Directed Association

A directed association in a UML class diagram represents a relationship between two classes where the association has a direction, indicating that one class is associated with another in a specific way.

- In a directed association, an arrowhead is added to the association line to indicate the direction of the relationship. The arrow points from the class that initiates the association to the class that is being targeted or affected by the association.
- Directed associations are used when the association has a specific flow or directionality, such as indicating which class is responsible for initiating the association or which class has a dependency on another.

Consider a scenario where a "Teacher" class is associated with a "Course" class in a university system. The directed association arrow may point from the "Teacher" class to the "Course" class, indicating that a teacher is associated with or teaches a specific course.

- The source class is the "Teacher" class. The "Teacher" class initiates the association by teaching a specific course.
- The target class is the "Course" class. The "Course" class is affected by the association as it is being taught by a specific teacher.

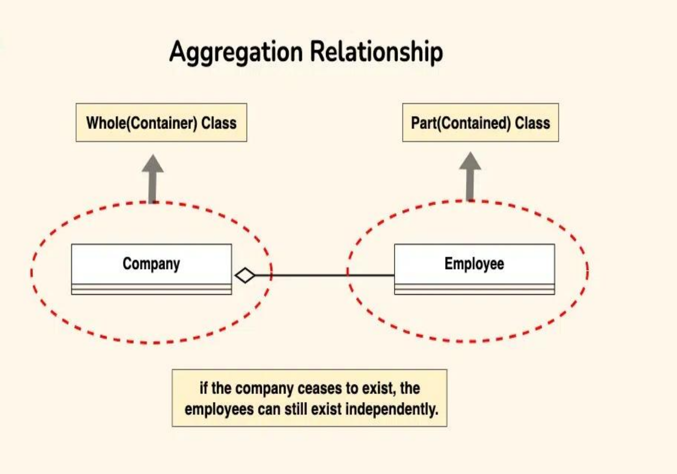

Directed Association Relationship
Source Class
Target Class
Teacher
Course
A "Teacher" class is associated with
A "Course" class in a university system

3. Aggregation

Aggregation is a specialized form of association that represents a "whole-part" relationship. It denotes a stronger relationship where one class (the whole) contains or is composed of another class (the part). Aggregation is represented by a diamond shape on the side of the whole class. In this kind of relationship, the child class can exist independently of its parent class.

Let's understand aggregation using an example:

The company can be considered as the whole, while the employees are the parts. Employees belong to the company, and the company can have multiple employees. However, if the company ceases to exist, the employees can still exist independently.



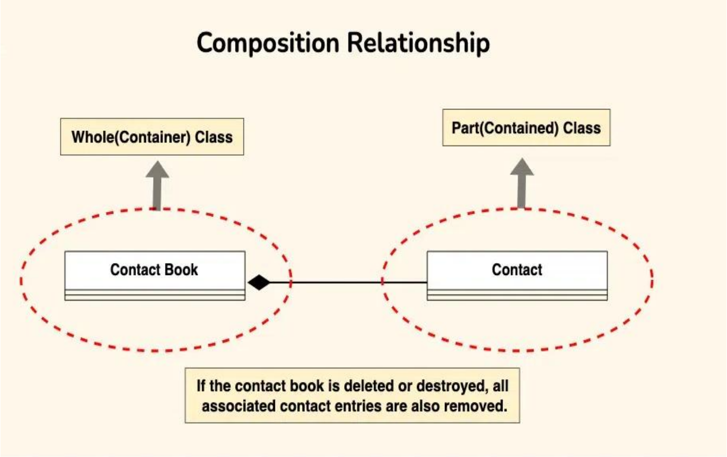
4. Composition

Composition is a stronger form of aggregation, indicating a more significant ownership or dependency relationship. In composition, the part class cannot exist independently of the whole class. Composition is represented by a filled diamond shape on the side of the whole class.

Let's understand Composition using an example:

Imagine a digital contact book application. The contact book is the whole, and each contact entry is a part. Each contact entry is fully owned and managed by the contact book. If the contact book is deleted or destroyed, all associated contact entries are also removed.

This illustrates composition because the existence of the contact entries depends entirely on the presence of the contact book. Without the contact book, the individual contact entries lose their meaning and cannot exist on their own.

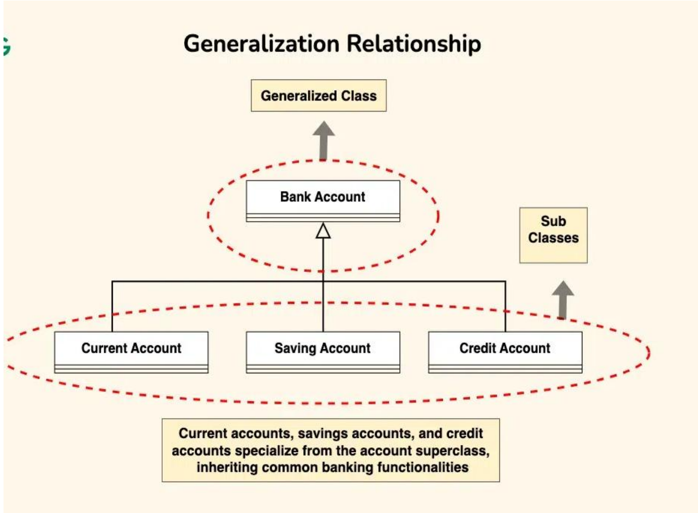

Composition Relationship
Whole(Container) Class
Part(Contained) Class
Contact Book
Contact
If the contact book is deleted or destroyed, all associated contact entries are also removed.

5. Generalization(Inheritance)

Inheritance represents an "is-a" relationship between classes, where one class (the subclass or child) inherits the properties and behaviors of another class (the superclass or parent). Inheritance is depicted by a solid line with a closed, hollow arrowhead pointing from the subclass to the superclass.

In the example of bank accounts, we can use generalization to represent different types of accounts such as current accounts, savings accounts, and credit accounts.

The Bank Account class serves as the generalized representation of all types of bank accounts, while the subclasses (Current Account, Savings Account, Credit Account) represent specialized versions that inherit and extend the functionality of the base class.


Generalization Relationship
Generalized Class
Bank Account
Sub Classes
Current Account
Saving Account
Credit Account
Current accounts, savings accounts, and credit accounts specialize from the account superclass, inheriting common banking functionalities

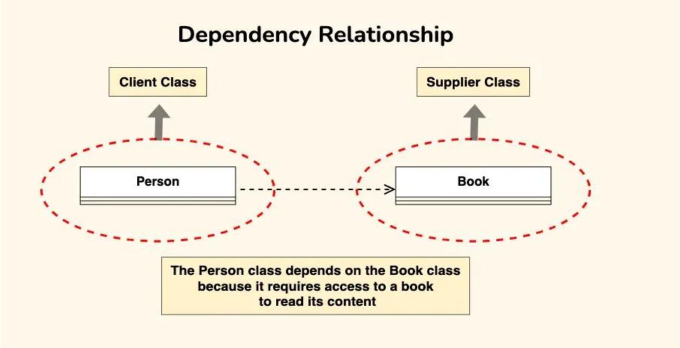
6. Dependency Relationship

A dependency exists between two classes when one class relies on another, but the relationship is not as strong as association or inheritance. It represents a more loosely coupled connection between classes. Dependencies are often depicted as a dashed arrow.

Let’s consider a scenario where a Person depends on a Book.

- **Person Class:** Represents an individual who reads a book. The Person class depends on the Book class to access and read the content.
- **Book Class:** Represents a book that contains content to be read by a person. The Book class is independent and can exist without the Person class.

The Person class depends on the Book class because it requires access to a book to read its content. However, the Book class does not depend on the Person class; it can exist independently and does not rely on the Person class for its functionality.


Dependency Relationship
Client Class
Supplier Class
Person
Book
The Person class depends on the Book class because it requires access to a book to read its content

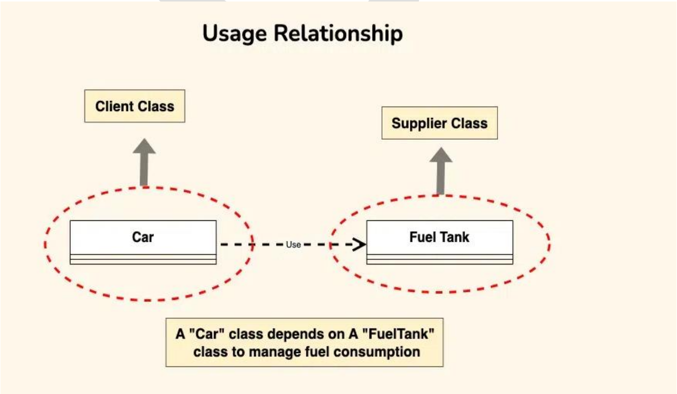
8. Usage(Dependency) Relationship

A usage dependency relationship in a UML class diagram indicates that one class (the client) utilizes or depends on another class (the supplier) to perform certain tasks or access certain functionality. The client class relies on the services provided by the supplier class but does not own or create instances of it.

- Usage dependencies represent a form of dependency where one class depends on another class to fulfill a specific need or requirement.
- The client class requires access to specific features or services provided by the supplier class.
- In UML class diagrams, usage dependencies are typically represented by a dashed arrowed line pointing from the client class to the supplier class.
- The arrow indicates the direction of the dependency, showing that the client class depends on the services provided by the supplier class.

Consider a scenario where a “Car” class depends on a “FuelTank” class to manage fuel consumption.

- The “Car” class may need to access methods or attributes of the “Fuel Tank” class to check the fuel level, refill fuel, or monitor fuel consumption.
- In this case, the “Car” class has a usage dependency on the “Fuel Tank” class because it utilizes its services to perform certain tasks related to fuel management.


Usage Relationship
Client Class
Supplier Class
Car
Use
Fuel Tank
A "Car" class depends on A "FuelTank" class to manage fuel consumption

Purpose of Class Diagrams

The main purpose of using class diagrams is:

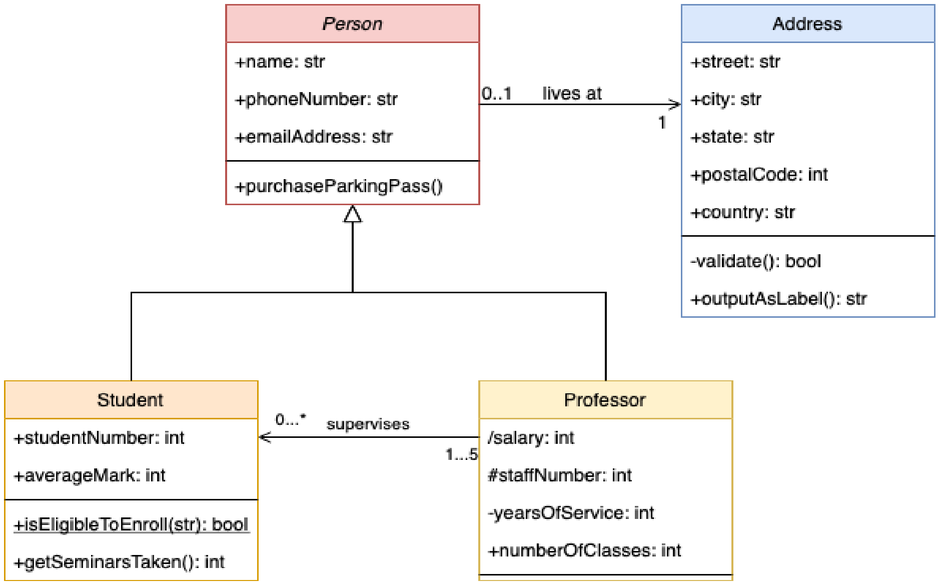
- This is the only UML that can appropriately depict various aspects of the OOPs concept.
- Proper design and analysis of applications can be faster and efficient.
- It is the base for deployment and component diagram.

- It incorporates forward and reverse engineering.

Benefits of Class Diagrams

- **Modeling Class Structure:**
  - Class diagrams help in modeling the structure of a system by representing classes and their attributes, methods, and relationships.
  - This provides a clear and organized view of the system's architecture.
- **Understanding Relationships:**
  - Class diagrams depict relationships between classes, such as associations, aggregations, compositions, inheritance, and dependencies.
  - This helps stakeholders, including developers, designers, and business analysts, understand how different components of the system are connected.
- **Communication:**
  - Class diagrams serve as a communication tool among team members and stakeholders. They provide a visual and standardized representation that can be easily understood by both technical and non-technical audiences.
- **Blueprint for Implementation:**
  - Class diagrams serve as a blueprint for software implementation. They guide developers in writing code by illustrating the classes, their attributes, methods, and the relationships between them.
  - This can help ensure consistency between the design and the actual implementation.
- **Code Generation:**
  - Some software development tools and frameworks support code generation from class diagrams.
  - Developers can generate a significant portion of the code from the visual representation, reducing the chances of manual errors and saving development time.
- **Identifying Abstractions and Encapsulation:**
  - Class diagrams encourage the identification of abstractions and the encapsulation of data and behavior within classes.
  - This supports the principles of object-oriented design, such as modularity and information hiding.

Example:



Object Diagrams

Because object diagrams are formed from class diagrams, they rely on them.

An object diagram represents a class diagram. Class and object diagrams have identical fundamental notions. Object diagrams also depict a system's static perspective, but this view is a snapshot of the system at a specific point in time.

Object diagrams are used to represent a collection of things and their relationships.

Purpose of Object Diagrams

A diagram's purpose must be well understood before being used in practice. Object diagrams serve the same goals as class diagrams.

On the other hand, a class diagram illustrates an abstract model made up of classes and their relationships. On the other hand, an object diagram shows an actual instance at a specific time.

It indicates that the object diagram is more accurate than the actual system behavior. The goal is to capture a system's static view at a specific point in time.

The object diagram's purpose can be summarized as follows:

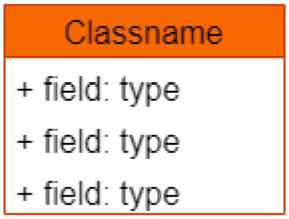
- Forward and reverse engineering.
- Object relationships of a system
- A static view of interaction.
- Understand object behavior and their relationship from a practical perspective

Notations Used in Object Diagram

Objects or Instance specifications

The object we produce when we instantiate a classifier in a system represents an entity that already exists in the system. Multiple instance specifications can be used to indicate changes in an object across time. In an Object Diagram, an object is represented by a rectangle. In an object diagram, one object is usually related to other objects.

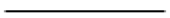
Object Diagram



Links

A link is a symbol that represents a connection between two objects.

Notation for a Link

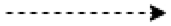


For each end of the link, we represent the number of participants. A relationship between two classifiers is referred to as an association. A relationship between two instance specifications or objects is defined by the term link. A solid line is used to depict a connection between two items.

Dependency Relationships

A dependence relationship is used to illustrate when one element is dependent on another.

Notation for Dependency Relationship



Dependency connections are used in class diagrams, component diagrams, deployment diagrams, and object diagrams. The relationship between dependent and independent components in a system is represented by dependence. Any change in one element's definition or structure may result in modifications in the other. A unidirectional relationship exists between two items.

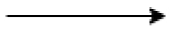
Keywords (sometimes enclosed in angular brackets) are used to specify different sorts of dependency connections.

The types of dependency relationships utilized in UML are abstraction, binding, realization, substitution, and usage.

Association

A reference relationship between two objects is called an association (or classes).

Notation for Abstraction

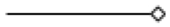


An association occurs when one object makes use of another. When one item refers to members of another object, we utilize association. The association might be one-way or bi-directional. The association is represented by an arrow.

Aggregation

A "has a" relationship is represented by aggregation.

Notation for Aggregation

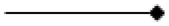


A special type of association is aggregation. Aggregation is more specific than an ordinary association because it is based on a connection. It is a representation of a part-whole or part-of relationship. It's similar to a parent-child connection, except it's not inherited. When the lifecycle of enclosed items is not substantially dependent on the lifecycle of container objects, aggregation occurs.

Composition

Composition is a sort of relationship in which one child cannot exist without the other.

Notation for Composition



Composition is a sort of relationship that is distinct from others. It's akin to a parent-child connection, but it's not the same as an inheritance.

Drawing an Object Diagram

An object diagram is a type of class diagram, as we've just described. It suggests that an object diagram is made up of examples of the items in a class diagram.
As a result, both diagrams are made up of the same basic pieces, but they are represented in different ways. The elements in the class diagram are abstract to represent the blueprint, whereas the elements in the object diagram are concrete to represent the real-world object.

The number of class diagrams available to capture a specific system is restricted. If we consider object diagrams, however, we can have an infinite number of instances, each of which is unique. Only those events that have an influence on the system are taken into account.

From the above discussion, it is evident that a single object diagram cannot capture all of the required instances, or to put it another way, a single object diagram cannot specify all of a system's objects.

As a result, the answer is to examine the system and determine which instances contain critical data and associations. Second, we should solely think about the scenarios that will cover the functionality. Third, because the number of instances is limitless, we are required to do some optimization.

Before drawing an object diagram, keep the following points in mind and make sure you understand them completely:

- Object diagrams are made up of various objects.
- The object diagram's link is used to connect things.
- An object diagram is made up of two elements: objects and linkages.

Following that, make the following decisions before beginning to design the diagram:

- The objective of the object diagram should be indicated by a meaningful name.
- It's time to figure out what is most essential.
- The relationship between things should be clarified.
- To include in the object diagram, the values of various elements must be captured.
- At times where further clarity is required, provide proper notes.

Example of Object Diagram

An object diagram is depicted in the diagram below. It reflects the Order management system explained in the Class Diagram chapter. The diagram below depicts the system at a specific point in time when it was purchased. It contains the following items.

- Customer
- Order
- SpecialOrder
- NormalOrder

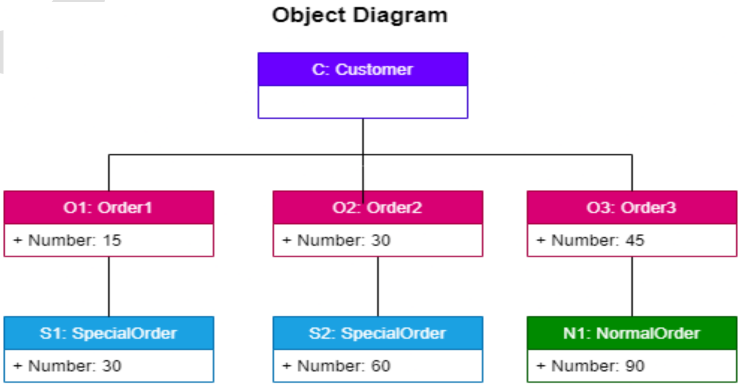
Three order objects are now connected with the customer object (C) (O1, O2, and O3). These order items are linked to both special and regular order objects (S1, S2, and N1).

For the time period in question, the customer has three orders with varying numbers (15, 30, and 45).

In the future, the customer may increase the number of orders, and the object diagram will reflect this. If you look at the order, special order, and normal order objects, you'll notice that they all have values.

The values for orders are 15, 30, and 45, indicating that the objects have these values during a specific time (here, the time when the purchase is made is considered the instant) when the instance is captured.

The same may be said for special order and normal order objects with orders of 30, 60, and 90. If a different time of purchase is taken into account, these figures will alter.
The following object diagram was created with all of the points in mind, as mentioned above.



Use case Diagram

What is a use case diagram?

In the Unified Modeling Language (UML), a use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system.
To build one, you'll use a set of specialized symbols and connectors. An effective use case diagram can help your team discuss and represent:

- Scenarios in which your system or application interacts with people, organizations, or external systems
- Goals that your system or application helps those entities (known as actors) achieve
- The scope of your system

When to apply use case diagrams

A use case diagram doesn't go into a lot of detail—for example, don't expect it to model the order in which steps are performed.
Instead, a proper use case diagram depicts a high-level overview of the relationship between use cases, actors, and systems.
Experts recommend that use case diagrams be used to supplement a more descriptive textual use case.

UML is the modeling toolkit that you can use to build your diagrams. Use cases are represented with a labeled oval shape.

Stick figures represent actors in the process, and the actor's participation in the system is modeled with a line between the actor and use case.

To depict the system boundary, draw a box around the use case itself.
UML use case diagrams are ideal for:

- Representing the goals of system-user interactions
- Defining and organizing functional requirements in a system
- Specifying the context and requirements of a system
- Modeling the basic flow of events in a use case

Use case diagram components

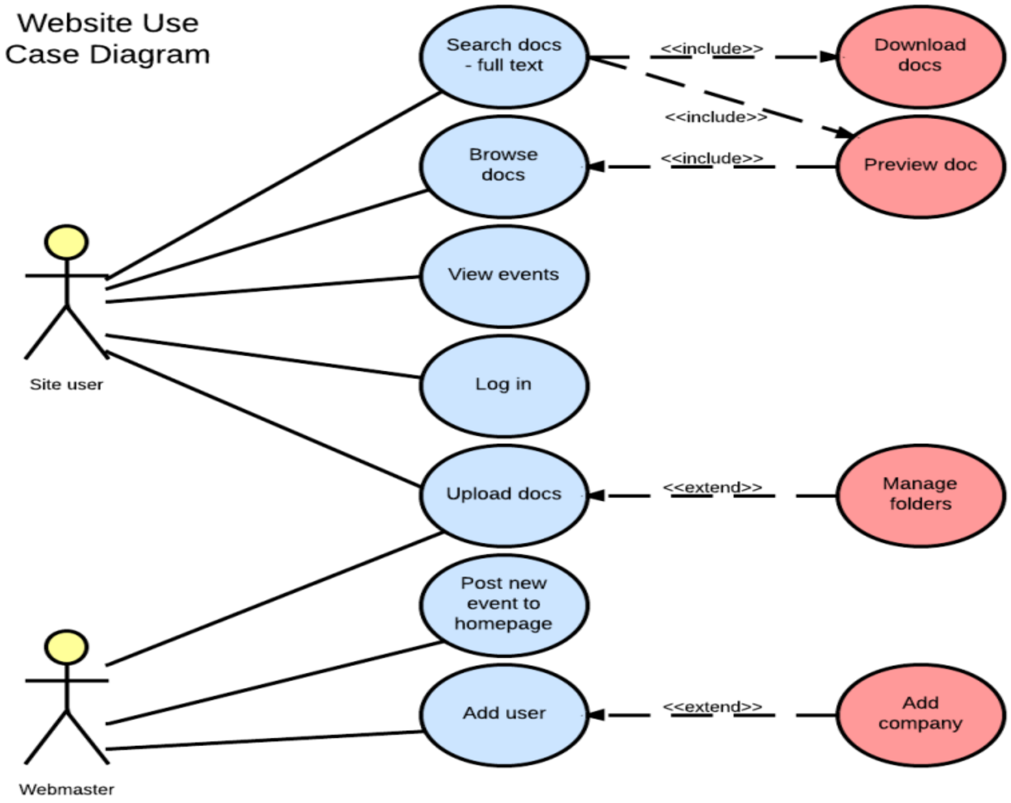
To answer the question, "What is a use case diagram?" you need to first understand its building blocks. Common components include:

- **Actors:** The users that interact with a system. An actor can be a person, an organization, or an outside system that interacts with your application or system. They must be external objects that produce or consume data.
- **System**: A specific sequence of actions and interactions between actors and the system. A system may also be referred to as a scenario.
- **Goals**: The end result of most use cases. A successful diagram should describe the activities and variants used to reach the goal.

Use case diagram symbols and notation

The notation for a use case diagram is pretty straightforward and doesn't involve as many types of symbols as other UML diagrams.

- **Use cases:** Horizontally shaped ovals that represent the different uses that a user might have.
- **Actors**: Stick figures that represent the people actually employing the use cases.
- **Associations**: A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.
- **System boundary boxes:** A box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system. For example, Psycho Killer is outside the scope of occupations in the chainsaw.
- **Packages**: A UML shape that allows you to put different elements into groups. Just as with component diagrams, these groupings are represented as file folders.



Activity Diagram

What is an activity diagram?

The Unified Modeling Language includes several subsets of diagrams, including structure diagrams, interaction diagrams, and behavior diagrams. Activity diagrams, along with use case and state machine diagrams, are considered behavior diagrams because they describe what must happen in the system being modeled.

Stakeholders have many issues to manage, so it's important to communicate with clarity and brevity. Activity diagrams help people on the business and development sides of an organization come together to understand the same process and behavior. You'll use a set of specialized symbols—including those used for starting, ending, merging, or receiving steps in the flow—to make an activity diagram, which we'll cover in more depth within this activity diagram guide.

Benefits of activity diagrams Activity diagrams present a number of benefits to users. Consider creating an activity diagram to:

- Demonstrate the logic of an algorithm.
- Describe the steps performed in a UML use case.
- Illustrate a business process or workflow between users and the system.
- Simplify and improve any process by clarifying complicated use cases.
- Model software architecture elements, such as method, function, and operation.

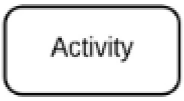
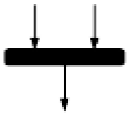
Basic components of an activity diagram

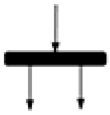
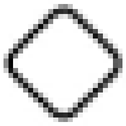
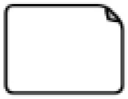
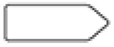
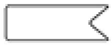
Before you begin making an activity diagram, you should first understand its makeup. Some of the most common components of an activity diagram include:

- **Action:** A step in the activity wherein the users or software perform a given task. In Lucid chart, actions are symbolized with round-edged rectangles.
- **Decision node:** A conditional branch in the flow that is represented by a diamond. It includes a single input and two or more outputs.
- **Control flows:** Another name for the connectors that show the flow between steps in the diagram.
- **Start node:** Symbolizes the beginning of the activity. The start node is represented by a black circle.
- **End node:** Represents the final step in the activity. The end node is represented by an outlined black circle.

Activity diagram symbols

These activity diagram shapes and symbols are some of the most common types you'll find in UML diagrams.

Symbol	Name	Description
●	Start symbol	Represents the beginning of a process or workflow in an activity diagram. It can be used by itself or with a note symbol that explains the starting point.
	Activity symbol	Indicates the activities that make up a modeled process. These symbols, which include short descriptions within the shape, are the main building blocks of an activity diagram.
→	Connector symbol	Shows the directional flow, or control flow, of the activity. An incoming arrow starts a step of an activity; once the step is completed, the flow continues with the outgoing arrow.
	Joint symbol/ Synchronization bar	Combines two concurrent activities and re-introduces them to a flow where only one activity occurs at a time. Represented with a thick vertical or horizontal line.

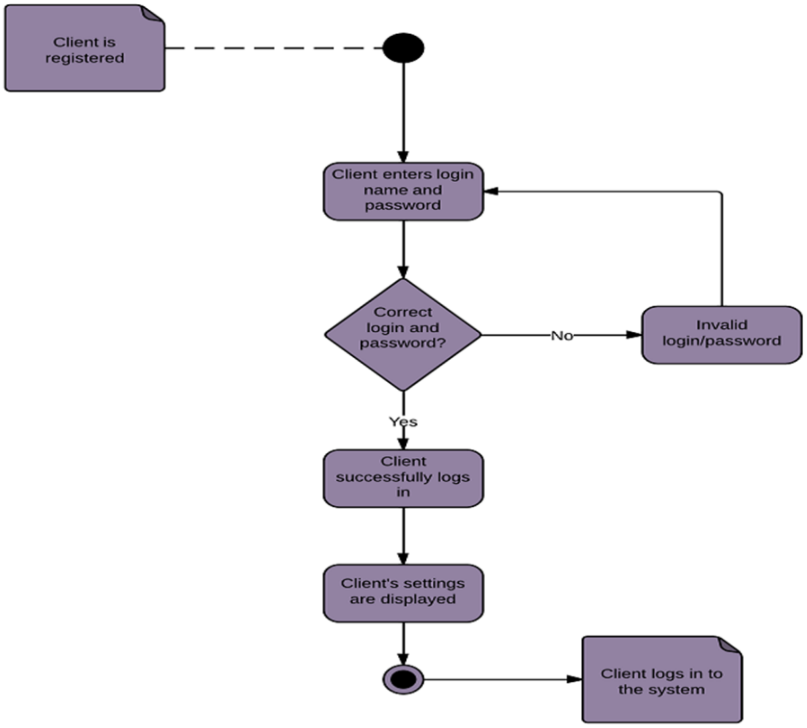
	Fork symbol	Splits a single activity flow into two concurrent activities. Symbolized with multiple arrowed lines from a join.
	Decision symbol	Represents a decision and always has at least two paths branching out with condition text to allow users to view options. This symbol represents the branching or merging of various flows with the symbol acting as a frame or container.
	Note symbol	Allows the diagram creators or collaborators to communicate additional messages that don't fit within the diagram itself. Leave notes for added clarity and specification.
	Send signal symbol	Indicates that a signal is being sent to a receiving activity.
	Receive signal symbol	Demonstrates the acceptance of an event. After the event is received, the flow that comes from this action is completed.

Ⓗ	Shallow history pseudo state symbol	Represents a transition that invokes the last active state.
▭	Option loop symbol	Allows the creator to model a repetitive sequence within the option loop symbol.
⊗	Flow final symbol	Represents the end of a specific process flow. This symbol shouldn't represent the end of all flows in an activity; in that instance, you would use the end symbol. The flow final symbol should be placed at the end of a process in a single activity flow.
[Condition]	Condition text	Placed next to a decision marker to let you know under what condition an activity flow should split off in that direction.
◉	End symbol	Marks the end state of an activity and represents the completion of all

Activity diagram for a login page

Many of the activities people want to accomplish online—checking email, managing finances, ordering clothes, etc.—require them to log into a website. This activity diagram shows the process of logging into a website, from entering a username and password to successfully logging in to the system.

It uses different container shapes for activities, decisions, and notes. Lucid chart is the ideal tool for creating any kind of UML flowchart, whether it's an activity diagram, a use case diagram, or a component diagram. Lucid chart offers in-editor collaboration tools and instant web publishing so you can demonstrate the functionality of your system to others.


Client is registered
Client enters login name and password
Correct login and password?
No
Invalid login/password
Yes
Client successfully logs in
Client's settings are displayed
Client logs in to the system

- **Case Study**

  Library Management system

  Class diagram

<u>Classes of Library Management System :</u>

- Library Management System class – It manages all operations of the Library Management System. It is the central part of the organization for which software is being designed.
- User Class – It manages all operations of the user.
- Librarian Class – It manages all operations of Librarian.
- Book Class – It manages all operations of books. It is the basic building block of a system.

- Account Class – It manages all operations of an account.
- Library database Class – It manages all operations of the library database.
- Staff Class – It manages all operations of staff.
- Student Class – It manages all operations of student.

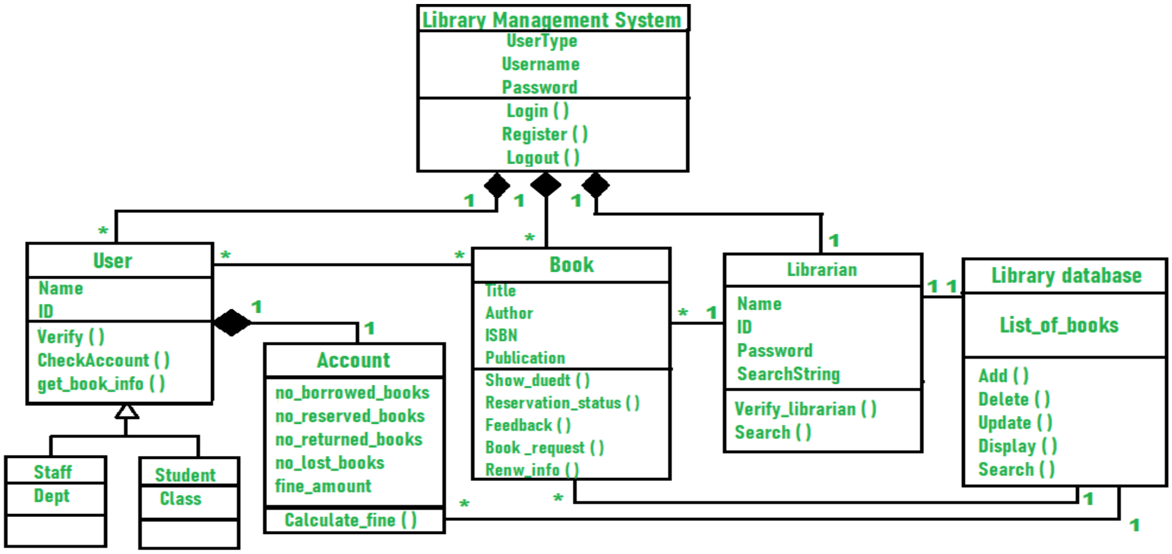
Attributes of Library Management System :

- Library Management System Attributes – UserType, Username, Password
- User Attributes – Name, Id
- Librarian Attributes – Name, Id, Password, SearchString
- Book Attributes – Title, Author, ISBN, Publication
- Account Attributes – no_borrowed_books, no_reserved_books, no_returned_books, no_lost_books fine_amount
- Library database Attributes – List_of_books
- Staff Class Attributes – Dept
- Student Class Attributes – Class

Methods of Library Management System :

- Library Management System Methods – Login(), Register(), Logout()
- User Methods – Verify(), CheckAccount(), get_book_info()
- Librarian Methods – Verify_librarian(), Search()
- Book Methods – Show_duedt(), Reservation_status(), Feedback(), Book_request(), Renew_info()
- Account Methods – Calculate_fine()
- Library database Methods – Add(), Delete(), Update(), Display(), Search()

Class Diagram of Library Management System :



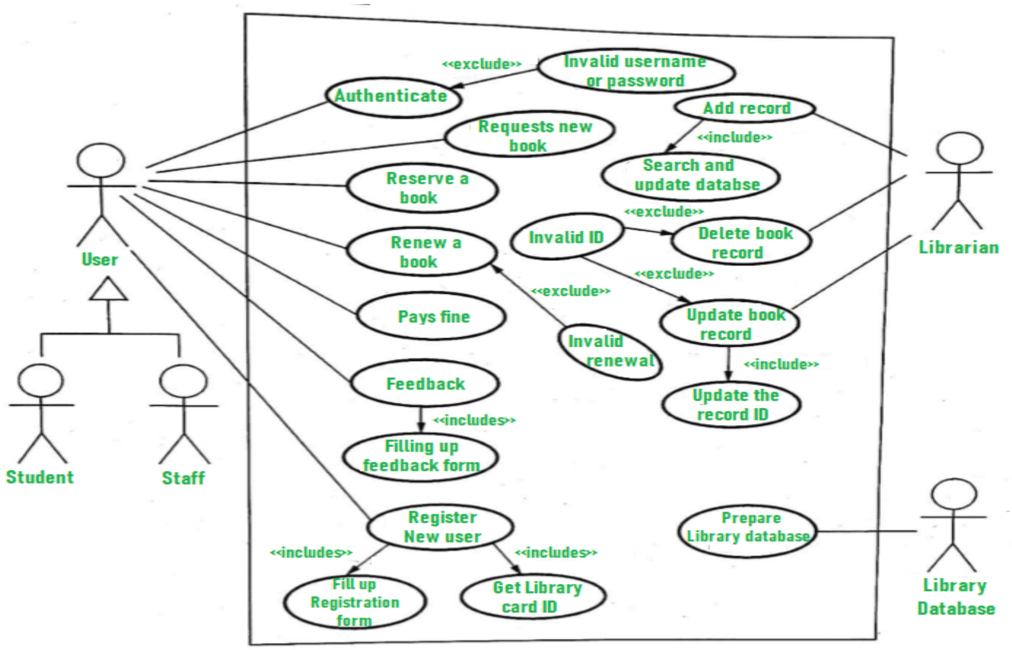
CLASS DIAGRAM FOR LIBRARY MANAGEMENT SYSTEM

Use Case Diagram for Library Management System

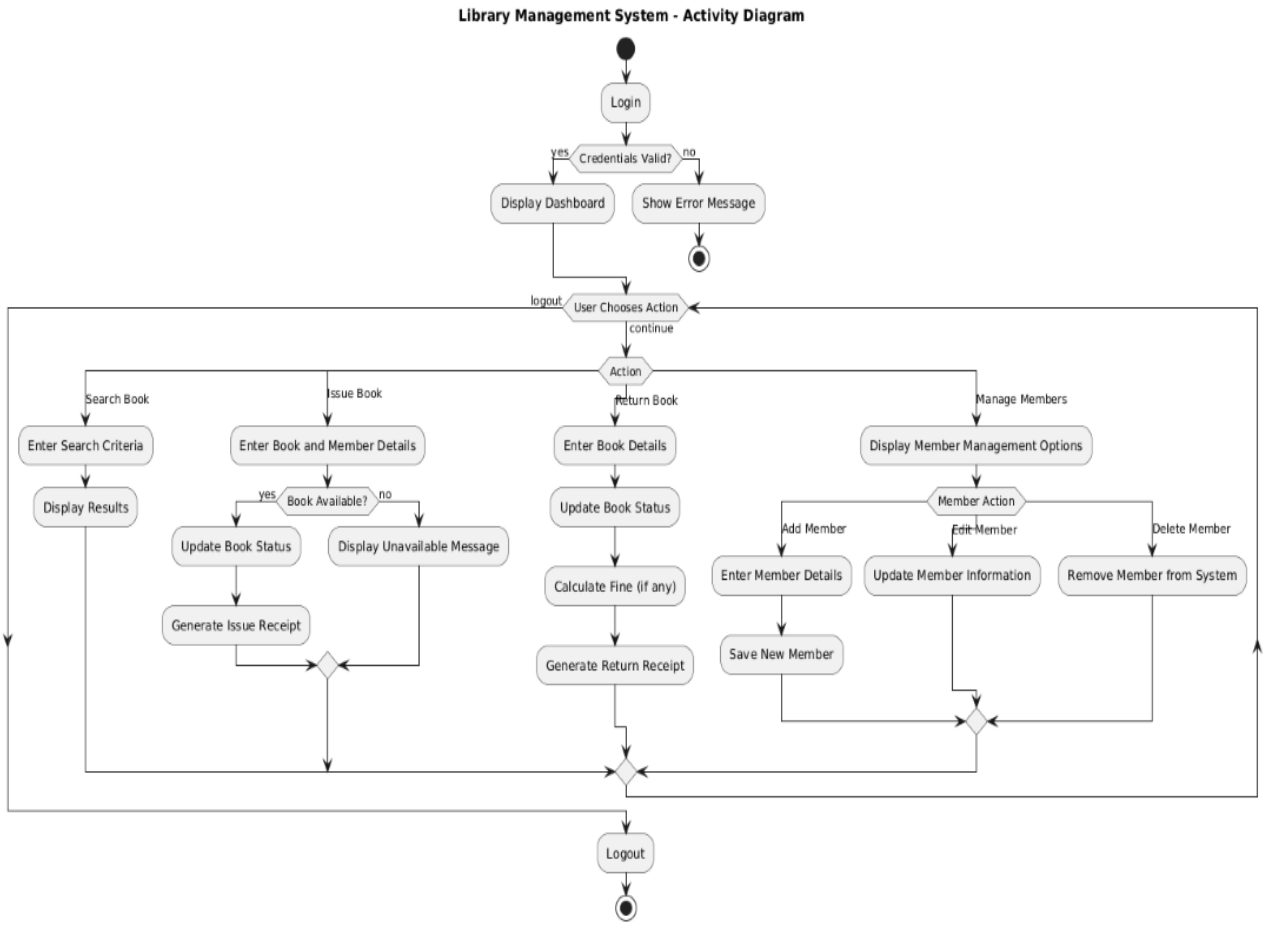
Let's visually map out the relationships and interactions. Below is the textual description of what the diagram would look like:

1. **Actors:**
   - User (Staff or Student)
   - Librarian
2. **Use Cases:**
   - Register New User
   - Issue Library Card
   - Request New Book
   - Reserve Book
   - Renew Book
   - Pay Fine
   - Fill Feedback Form
   - Manage Records
   - Delete Records
   - Update Database
3. **System Boundary:**
   - The system boundary will encompass all the use cases mentioned above.

Below is the use case diagram of a Library Management System



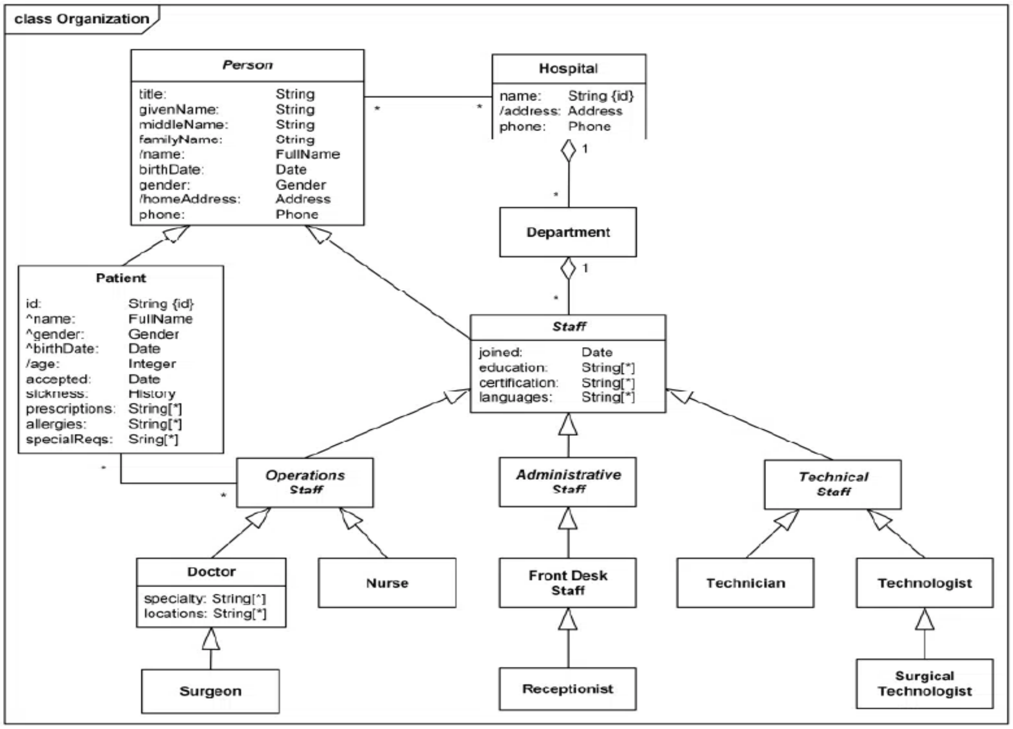
Activity Diagram of Library Management System



➢ Case study for hospital management system

Class Diagram of Hospital Management System

Class Diagram for Hospital Management System simply describes structure of Hospital Management System class, attributes, methods or operations, relationship among objects.



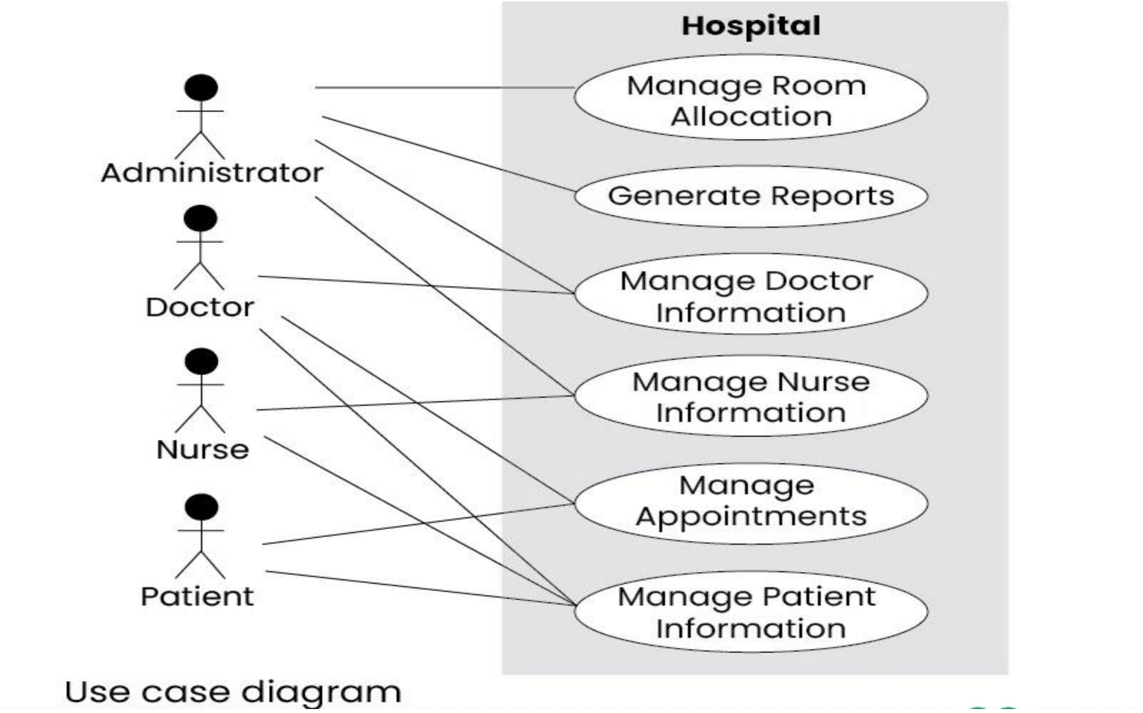
For a Hospital Management System (HMS), the class diagram and its characteristics would typically include:

1. **Classes**: Representing entities such as Patient, Doctor, Nurse, Administrator, Appointment, MedicalRecord, Department, and Billing.
2. **Attributes**: Each class would have attributes representing its properties. For example, the Patient class might have attributes like patientID, name, gender, dateOfBirth, and contactDetails.
3. **Operations (Methods)**: These would define the behaviors associated with each class. For instance, the Appointment class might have methods like scheduleAppointment(), cancelAppointment(), and rescheduleAppointment().
4. **Associations**: Relationships between classes would be depicted to show how they are connected. For instance, an association between Patient and Doctor classes would show that a patient can be associated with one or more doctors.
5. **Multiplicity**: Multiplicity would specify how many instances of one class are associated with instances of another class. For example, a Doctor can have multiple patients, indicating a one-to-many relationship.

6. **Inheritance (Generalization)**: If there are common attributes or methods shared between classes, inheritance can be used to depict this relationship. For example, Doctor and Nurse classes might inherit from a common super class called Healthcare Professional.
7. **Composition and Aggregation**: These relationships would show how one class contains or is composed of another class. For example, a Hospital class may have a composition relationship with the Department class, indicating that a hospital consists of multiple departments.
8. **Dependency**: This would show when one class relies on another class, usually through method parameters or return types. For example, the Billing class may have a dependency on the Patient class to retrieve patient information for billing purposes.

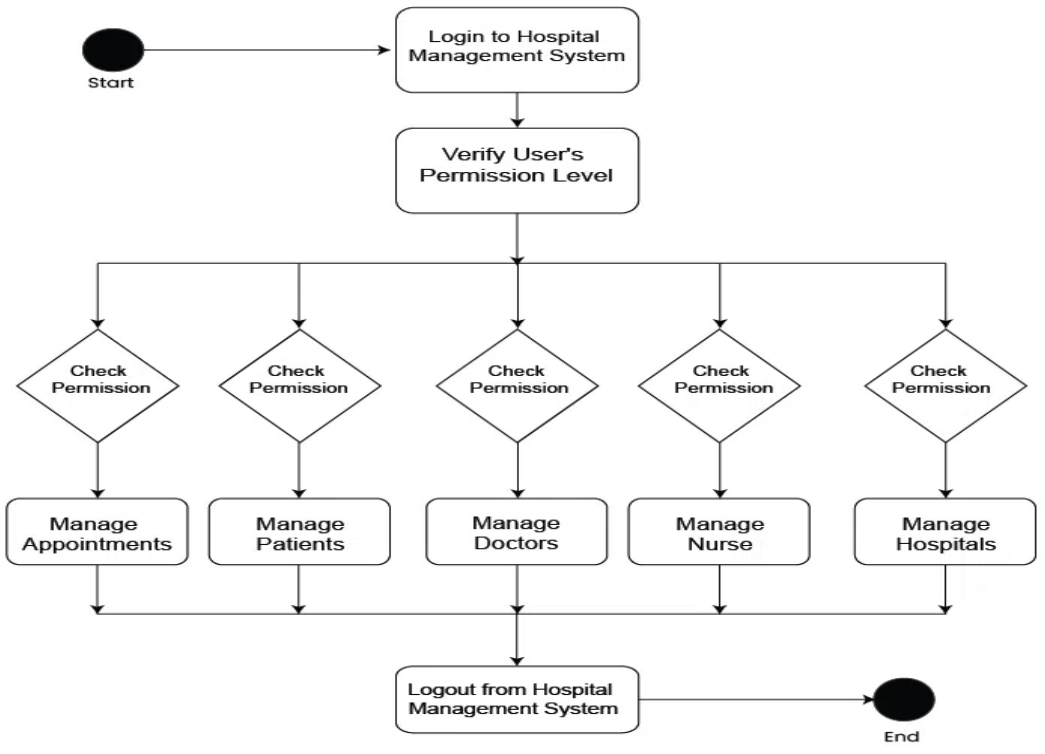
Use case Diagram for Hospital Management System:

A Use Case Diagram for a Hospital Management System (HMS) depicts the interactions between actors (users or external systems) and the system to achieve specific goals. Here's a simplified version of a Use Case Diagram for an HMS:



Activity Diagram of Hospital Management System

This activity diagram outlines the main activities and interactions within the Hospital Management System, including booking appointments, patient check-in, accessing medical records, billing, inventory management, staff management, and report generation. Each activity represents a specific task or function performed by users or the system.



Activity Diagram of Hospital Management System

Activity Diagram of Hospital Management System

1. **User Interaction:**
    1. Book Appointment
    2. Check in
    3. View Medical Records
    4. Pay Bills
    5. Manage Inventory
    6. Manage Staff
    7. Generate Reports
    8. Log Out
2. **Book Appointment:**
    1. User selects “Book Appointment.”
    2. System displays available doctors and time slots.
    3. User selects a doctor and preferred time.
    4. System confirms the appointment booking.
3. **Check in:**
    1. Patient arrives at the hospital.
    2. Receptionist greets the patient and verifies their appointment.
    3. Receptionist checks the patient in and assigns a queue number.
4. **View Medical Records:**
    1. User selects “View Medical Records.”

   2. System prompts the user to enter a patient ID or name.
   3. System retrieves and displays the patient’s medical history.

5. **Pay Bills:**
   1. User selects “Pay Bills.”
   2. System displays a list of outstanding bills for the user.
   3. User selects the bill(s) to pay and enters payment details.
   4. System processes the payment and updates the billing records.
6. **Manage Inventory:**
   1. Authorized staff member selects “Manage Inventory.”
   2. System displays options to add, update, or remove items from inventory.
   3. Staff members perform the desired inventory management tasks.
   4. System updates the inventory database accordingly.
7. **Manage Staff:**
   1. Authorized administrator selects “Manage Staff.”
   2. System displays options to add, update, or remove staff members.
   3. Administrator performs the desired staff management tasks.
   4. System updates the staff database accordingly.
8. **Generate Reports:**
   1. Authorized user selects “Generate Reports.”
   2. System provides options to generate various reports such as patient statistics, financial summaries, etc.
   3. User selects the type of report to generate.
   4. System generates the report and displays it to the user.